# Algoritmusok és adatszerkezetek gyakorlat – 06

# Adatszerkezetek

## Tömb

- Ugyanolyan típusú elemeket tárol
- A mérete előre definiált kell legyen és nem lehet megváltoztatni futás során
- Legyen $n$ a tömb mérete. Ekkor:
  - **Elérési idő:** $O(1)$, mert az elemek egymás után folyamatosan tárolódnak a memóriában
  - **Beszúrás:** $O(n)$ legrosszabb esetben, ha a tömb elejére akarunk beszúrni és minden eddigi elemet arrébb kell rakni.
  - **Törlés:** $O(n)$ legrosszabb esetben, ha a tömb elejéről törlünk és minden további elemet egyel előrébb kell rakni
  - **Keresés:** $O(\log n)$, ha rendezett a tömb (bináris keresés) és $O(n)$, ha nem (szekvenciális keresés)

## Láncolt listák

- Minden elem egy adatból és egy (vagy több) mutatóból áll
- A mérete futás során módosítható
- Típusai:
  1. Egyirányú lista: minden elem egy adatot és egy rákövetkező elemre mutató pointert/referenciát tárol. Az utolsó elem pedig egy NULL-ra mutat. Például: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow NULL$
  2. Kétirányú lista: minden elem két pointert/referenciát tárol az adat mellett. Egyet a rákövetkező elemre, egy másikat a megelőzőre.

     Előnye, hogy mindkét irányban bejárható és törlésnél nem kell tudnunk a megelőző csúcs címét.

     Az első és utolsó eleme is NULL. Például: $NULL \leftarrow 1 \leftrightarrow 2 \leftrightarrow 3 \rightarrow NULL$
  3. Körben láncolt lista: az összes elem egy körbe van kötve. Nincs NULL elem a „végén”, az utolsó csúcs rákövetkező pointere az első elemre mutat. Lehet egyirányú és kétirányú láncolt is.

     Előnye, hogy bármelyik elemet kijelölhetjük kezdő elemnek.
- Legyen $n$ a lista hossza. Ekkor:
  - **Elérési idő:** $O(n)$, mert a lista elejétől végig kell keresni
  - **Beszúrás:** $O(1)$, ha már azon a pozíción vagyunk, ami után be akarunk szúrni

- **Törlés:** $O(1)$, ha már a törölt elem pozícióján vagyunk és tudjuk a törölni kívánt csúcs megelőzőjének a címét (vagy ha a megelőző elemen vagyunk)
- **Keresés:** $O(n)$

**Verem**

- A verem egy LIFO (last in, first out) adatszerkezet
- Két műveletet támogat:
  - push: egy elemet hozzáadunk az eddigiekhez úgy, hogy a verem tetejére tesszük
  - pop: az utoljára beszúrt elemet veszi ki a veremből (a tetejéről)
- Legyen $n$ a verem mérete. Ekkor:
  - **Elérési idő:** $O(1)$, de csak a verem tetején lévő elemet tudjuk elérni
  - **Beszúrás:** $O(1)$, mert mindig a tetejére pakolunk
  - **Törlés:** $O(1)$, de csak a tetején lévő elemet tudjuk törölni

**Sor**

- A sor egy FIFO (first in, first out) adatszerkezet
- Két műveletet támogat:
  - enqueue: egy új elemet adunk hozzá úgy, hogy a sor végére szúrjuk be
  - dequeue: az első elemet töröljük a sorból
- Legyen $n$ a sor mérete. Ekkor:
  - **Elérési idő:** $O(n)$ legrosszabb esetben
  - **Beszúrás:** $O(1)$
  - **Törlés:** $O(1)$

**Prioritási Sor**

- Absztrakt adatszerkezet, melyben az elemeket prioritásuk szerint tároljuk
- Három műveletet támogat:
  - insert: beszúr egy elemet
  - pop: kiveszi a legkisebb prioritású elemet
  - min: a legkisebb prioritású elemet adja vissza (pl. int-ek esetén minimumot)
- Legyen $n$ a PriSor mérete. Egy standard implementációban:
  - **Min:** $O(1)$
  - **Beszúrás:** $O(\log n)$
  - **Törlés:** $O(\log n)$

**1. Feladat** Adott a $3, 6, 10, 8, 1, 9, 7$ számsorozat, melyeket ebben a sorrendben tárolunk el. Adjuk meg milyen sorrendben vehetjük ki az elemeket verem, sor és prioritási sor adatszerkezet esetén!

**Megoldás**

**Verem:** Mivel egy LIFO adatszerkezetről van szó, így a legkésőbb berakott elem kerül ki legelőször. A sorrend tehát megfordul: $7, 9, 1, 8, 10, 6, 3$

**Sor:** ez egy FIFO adatszerkezet, tehát az jön ki először, amit legelőször tettünk be. A sorrend megmarad: $3, 6, 10, 8, 1, 9, 7$

**Prioritási sor:** az elemek rendezésre kerülnek az adatszerkezetben, tehát a sorrend: $1, 3, 6, 7, 8, 9, 10$

**2. Feladat** Szimuláljunk veremmel sort!

**Megoldás**

A sor egy FIFO adatszerkezet, ami azt jelenti, hogy az elem, amit elsőnek adtunk hozzá, elsőnek is kell kikerüljön.

Két műveletet kell támogatnunk:

- **enqueue:** a sorba művelethez elég egyetlen verem, amibe push művelettel eltároljuk az elemeket
- **dequeue:** az első elemet csak úgy tudjuk kiszedni a veremből, ha mindenkit kipakolunk pop művelettel. Mivel csak vermet használhatunk, így ehhez egy második veremre is szükség lesz, amibe át fogjuk rakni a kiszedett elemeket.

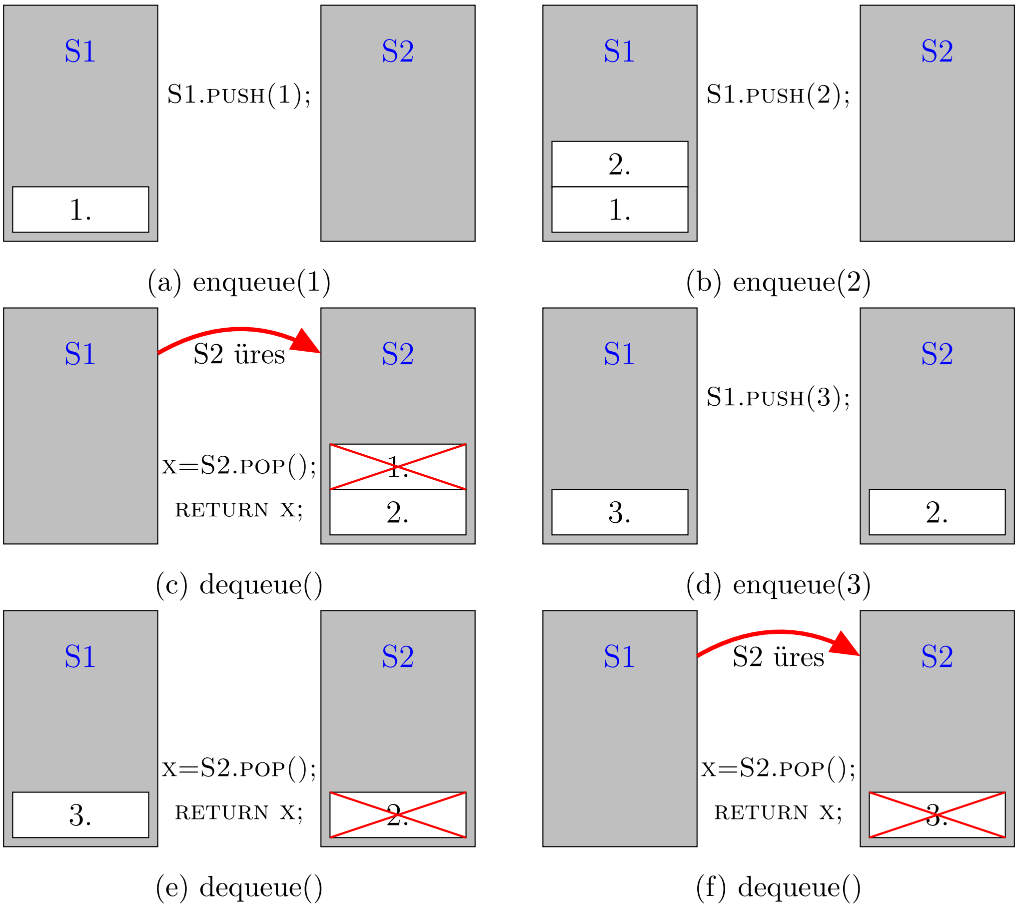
**Adat hozzáadása a sorhoz:**

Mivel a sorunk vége az első veremben, a sorunk eleje pedig a második veremben megfordított sorrendben van tárolva, egyszerűen az első veremhez push művelettel hozzáadjuk az elemet.

**Adat eltávolítása a sorból:**

Amikor a sorból kiveszünk egy elemet, az azt jelenti, hogy a legelső elemet kell kivennünk, ami bekerült a sorba. De ha egyszerűen csak az első veremből (S1) pop-pal veszünk ki elemet, az a legutoljára belerakottat adja vissza.

1. Ha a második verem (S2) üres, pakoljuk át bele az összes elemet az elsőből.
2. Vegyük ki S2 tetejéről az első elemet (mivel S2-ben épp a fordított sorrend van, így az épp az első elem lesz). Ha S2 üres, dobjunk hibát.




**1. Ábra.**: Példa az adatszerkezet működésére.

**3. Feladat** Adjunk meg egy olyan verem adatszerkezetet, amely támogatja a push és pop művelet mellett a minimum lekérdezését is $O(1)$ időben és plusz $O(n)$ tárban.

### Megoldás

Minden rész veremre el kell tároljuk a minimumot, nem csak az aktuális teljes veremre, így a „jegyezzük meg az aktuális minimumot” nem jó taktika.

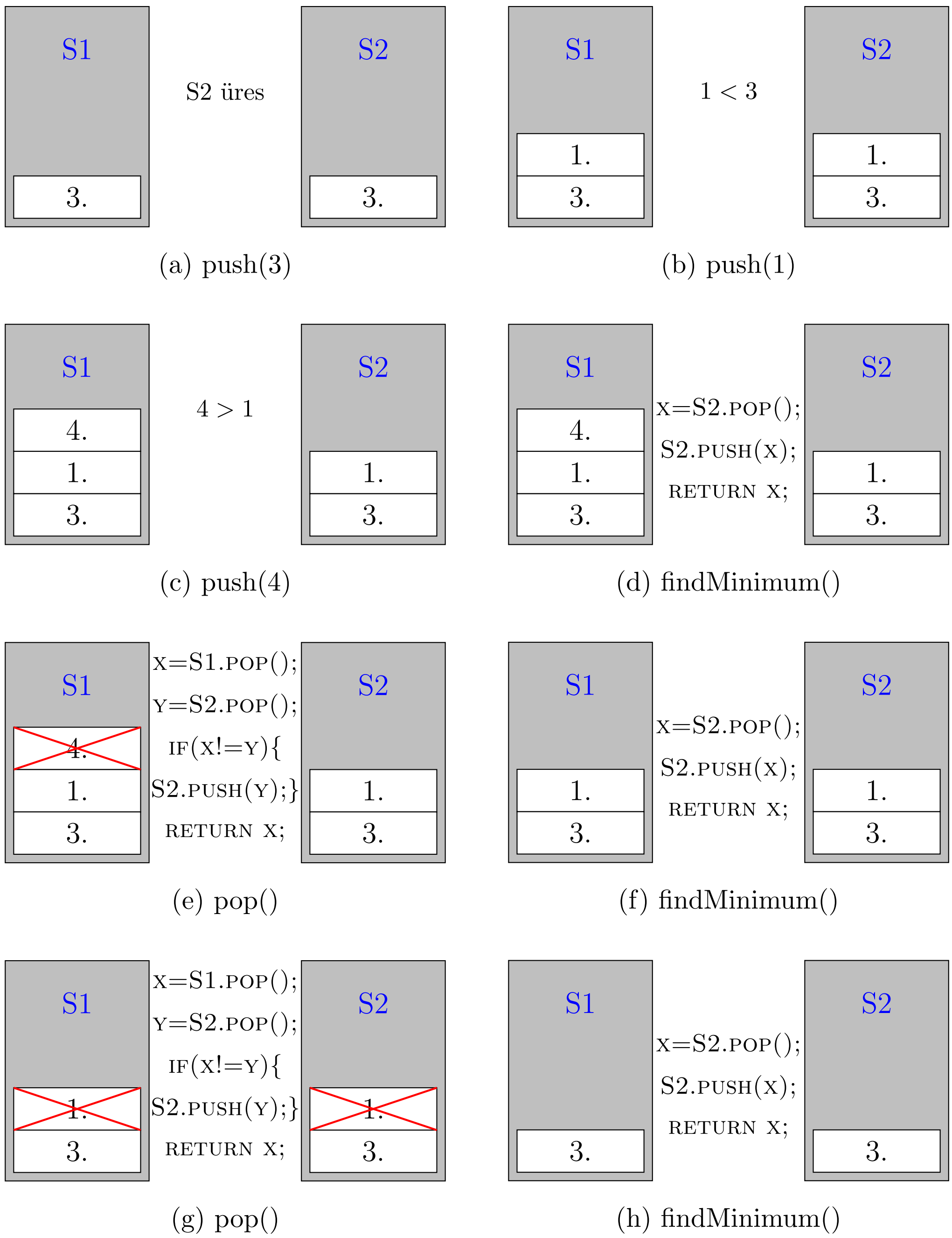
Használjunk két vermet. Az egyikben tároljuk ténylegesen az összes elemet (S1). A másik veremben csak a részvermek minimumát tároljuk (S2).

A műveletek:

- **Push:** rakjuk be az elemet az első verembe. Ha a második verem üres, rakjuk be oda is. Ha S2 nem üres, nézzük meg, hogy kisebb vagy egyenlő-e, mint a tetején lévő elem. Ha igen, adjuk hozzá ahhoz is.
- **Pop:** Vegyünk ki egy elemet az első veremből. Ha ez az elem ugyanaz, mint a minimumokat tároló verem tetején lévő elem, vegyük ki azt is.

- **FindMinimum:** egyszerűen adjuk vissza azt az elemet, amelyik a minimumokat tároló verem tetején van (de onnan ne szedjük ki).

Így minden műveletet $O(1)$ időben tudunk megvalósítani, és maximum még egyszer annyi tárat használunk, mint amekkora a teljes verem.



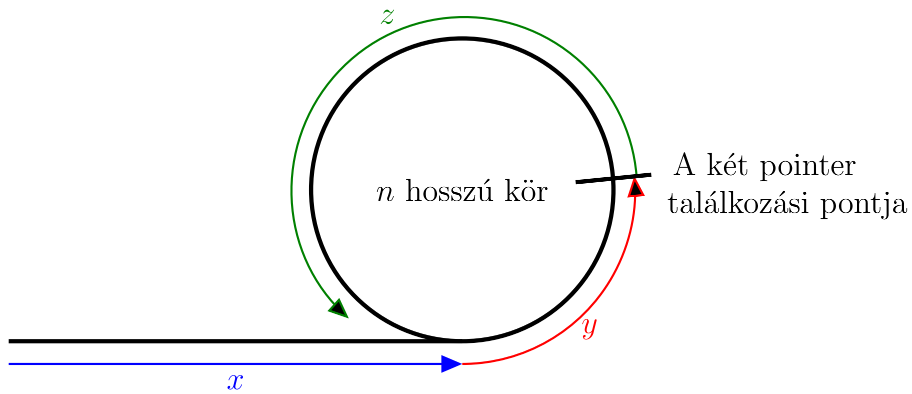

**2. Ábra.**: Példa az adatszerkezet működésére.

**4. Feladat** Hogyan tudjuk egy egyszeresen láncolt listában ellenőrizni, hogy van-e benne kör $O(n)$ időben és konstans tárban?

### Megoldás

A konstans tár azt jelenti, hogy nem tárolhatjuk el az elemeket máshol és nézhetjük meg, hogy már megtaláltuk-e, tehát csak pointereket tárolhatunk, amiket végigfuttathatunk a listán. Ebből egy nem lesz elég, mert hiába megyünk végig a listán, nem tudjuk eltárolni az addig látott elemeket.

Két pointer: mindkettőt a lista elejéről indítjuk. Az egyikkel minden iterációban egyet lépünk, a másikkal kettőt (vagy bármilyen más eltérő kombinációban). Ha a két pointer értéke bármelyik iterációban megegyezik (ugyanazon az elemen állnak), akkor van kör a láncolt listában (mert akkor a ,,gyorsabb” pointer ,,utolérte” a lassabbat, ami csak akkor lehet, ha egy körben vannak), ha pedig bármelyik mutató a lista végére ér, nincs benne kör (mivel egyszeresen láncolt, ha elérünk a végét jelző NULL elemig, az csak úgy lehet, hogy végig egyenesen haladtunk).




**3. Ábra.**: Ha van a listában kör, annak valamely pontján találkozni fognak a pointerek. Itt például míg a lassabb pointer az $x + y$ távot teszi meg, addig a gyorsabb az $x + y + z + y$-t.

Érdekesség: ha tudni akarod, hogyan lehet a minimum műveletet támogató vermet megvalósítani konstans tárral, kattints a nyuszira: